Optimizing Streams Applications

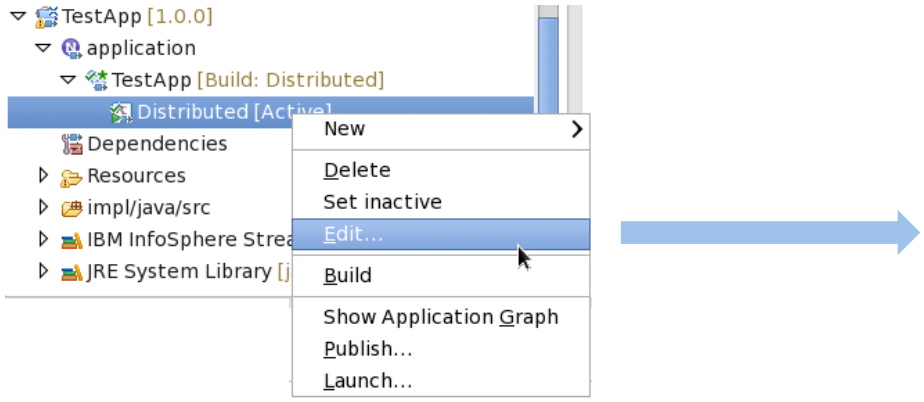
Scott Schneider

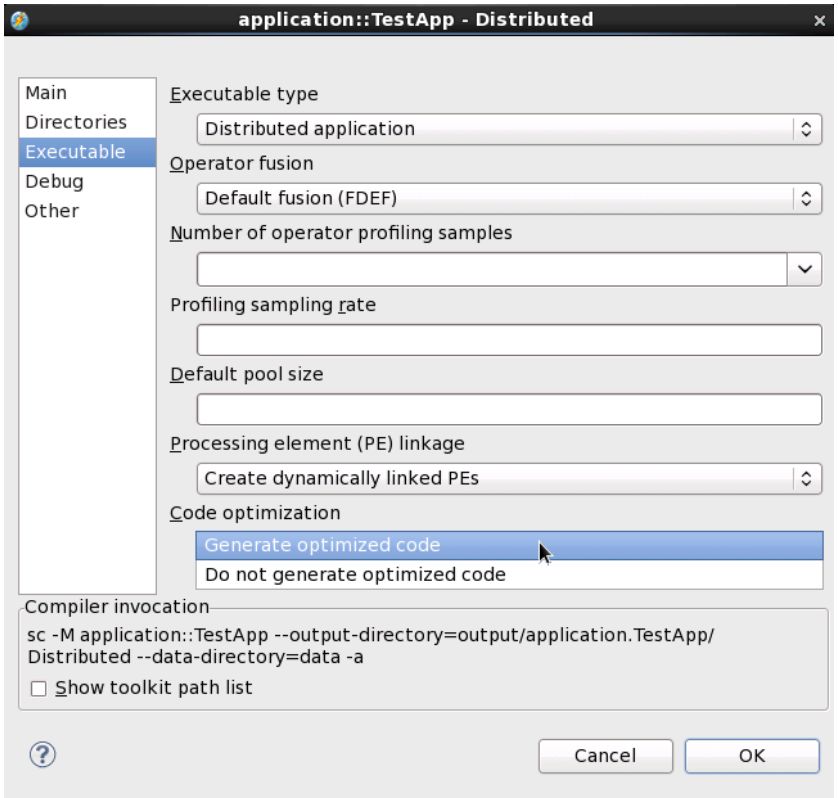
scott.a.s@us.ibm.com

version 1.0: April 4, 2014

Lesson 0

Compile with –a.





An Operator Graph

```
composite Main {
  type Entry = uint32 uid, rstring server, rstring msg;
       Summary = uint32 uid, int32 total;
  graph
   stream<Entry> Msgs = TCPSource() {
     param role: server;
           port: "http";

   }

   stream<Summary> Sums = Aggregate(Msgs) {
     window Msgs: tumbling, time(5), partitioned;
     param partitionBy: uid;
     output Sums: uid = Any(uid), total = Count();

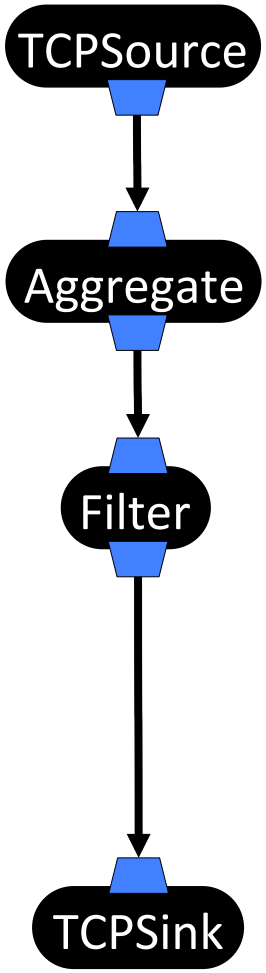
   }

   stream<Summary> Suspects = Filter(Sums) {
     param filter: total > 100;

   }

   () as Sink = TCPSink(Suspects) {
     param role: client;
           address: "suspects.acme.com";
          port: "http";

   }
}
```



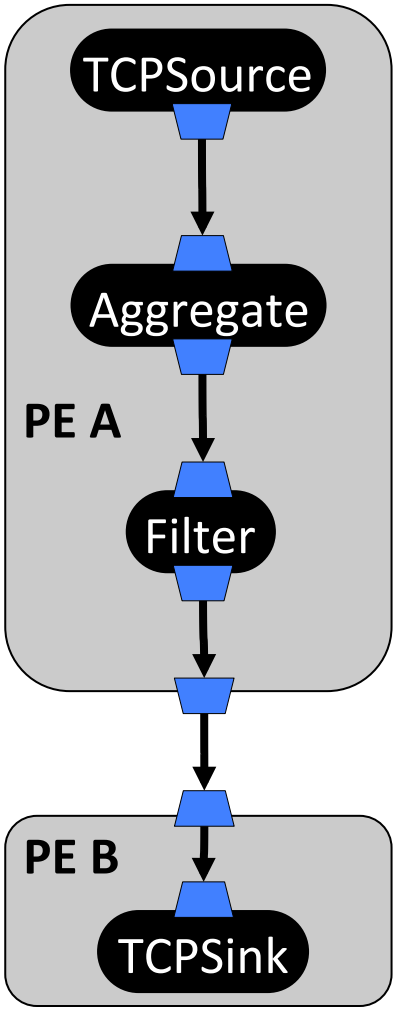
Fusing Operators into PEs

```
composite Main {
  type Entry = uint32 uid, rstring server, rstring msg;
       Summary = uint32 uid, int32 total;
  graph
   stream<Entry> Msgs = TCPSource() {
     param role: server;
           port: "http";
    config placement: partitionColocation("PE A");
   }

   stream<Summary> Sums = Aggregate(Msgs) {
     window Msgs: tumbling, time(5), partitioned;
     param partitionBy: uid;
     output Sums: uid = Any(uid), total = Count();
    config placement: partitionColocation("PE A");
   }

   stream<Summary> Suspects = Filter(Sums) {
     param filter: total > 100;
    config placement: partitionColocation("PE A");
   }

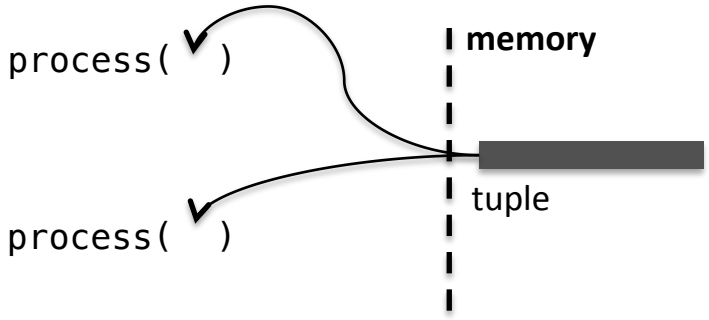
   () as Sink = TCPSink(Suspects) {
     param role: client;
           address: "suspects.acme.com";
           port: "http";
    config placement: partitionColocation("PE B");
   }
}
```



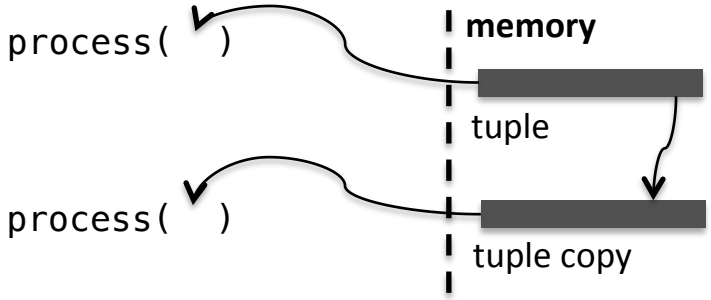
Tuple Transferring Costs Inside a PE

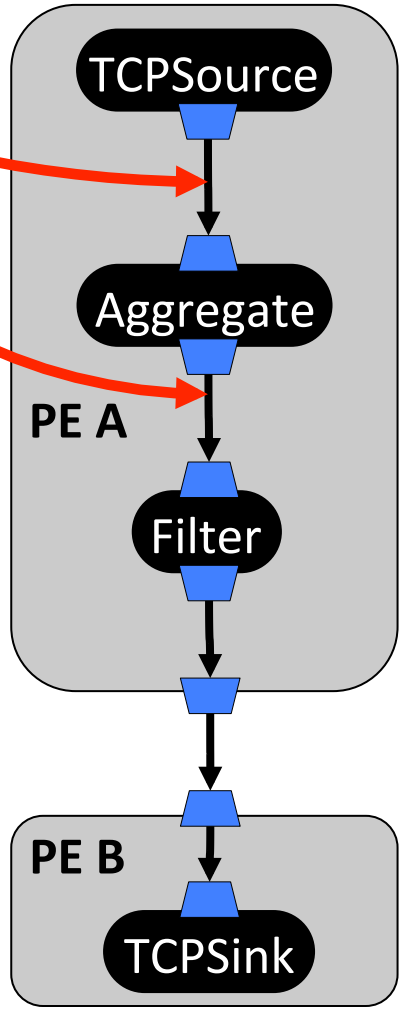
Operators inside of a PE communicate through function calls in 1 of 2 ways:

1. passing a reference (for non-threaded ports, and when downstream operator will not mutate the tuple):



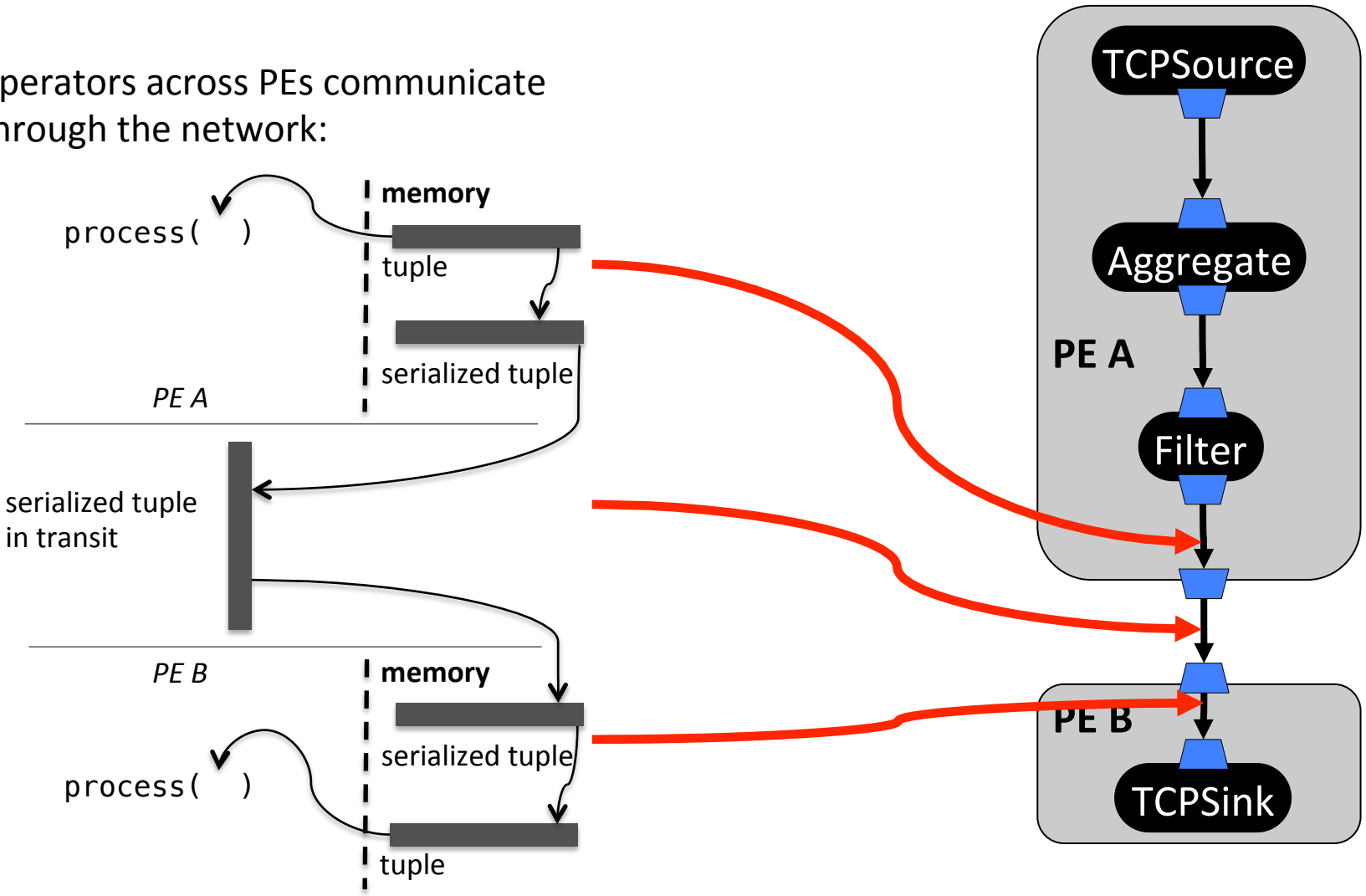
2. copying the tuple (if the downstream operator can mutate the tuple):





Tuple Transferring Costs Across PEs

operators across PEs communicate through the network:



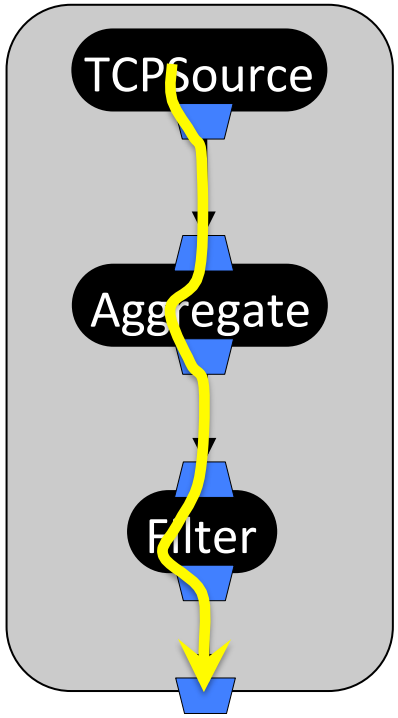
Lesson 1

Fuse operators into the same PE
to reduce communication costs.

Operator Execution Inside of a PE

In this PE, a thread that originates at the `TCPSource` will:

1. execute `TCPSource`
2. execute `Aggregate`
3. execute `Filter`
4. submit the tuple to the network
5. loop back to step 1



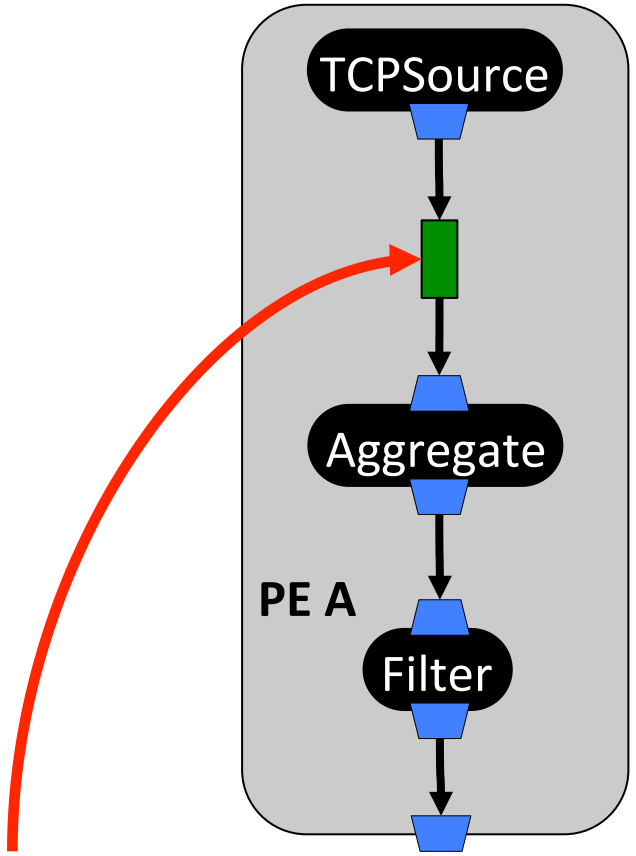
By fusing these operators into the same PE, we have minimized their communication cost, but we have lost **pipeline parallelism**. How do we get it back?

Threaded Ports

```
stream<Entry> Msgs = TCPSource() {
  param role: server;
        port: "http";
  config placement: partitionColocation("PE A");
}

stream<Summary> Sums = Aggregate(Msgs) {
  window Msgs: tumbling, time(5), partitioned;
  param partitionBy: uid;
  output Sums: uid = Any(uid), total = Count();
  config placement: partitionColocation("PE A");
         threadedPort: queue(Msgs, Sys.Wait);
}

stream<Summary> Suspects = Filter(Sums) {
  param filter: total > 100;
  config placement: partitionColocation("PE A");
}
```

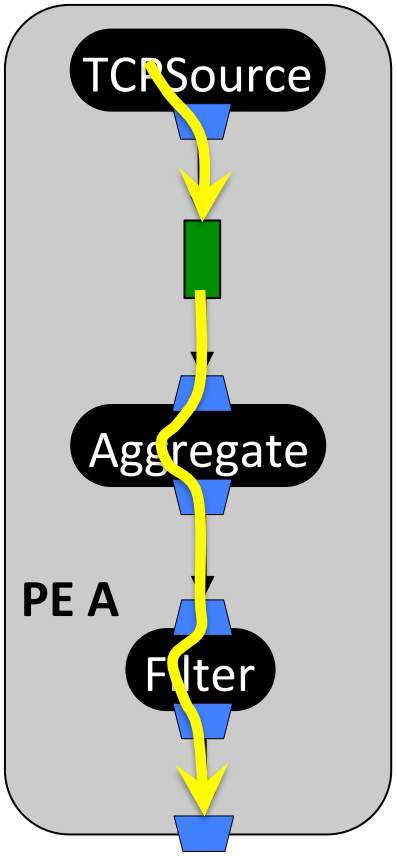


inserts a queue between operators

Execution With Threaded Ports

The thread that originates at the `TCPSource` will:

1. execute `TCPSource`
2. copy the tuple and place it in the threaded port queue
3. loop back to step 1



The thread that originates at the threaded port will:

1. pull a tuple from the threaded port queue
2. execute `Aggregate`
3. execute `Filter`
4. submit the tuple to the network
5. loop back to step 1

Both threads execute in **parallel**. By using a threaded port, we have introduced **pipeline paralleism** into a single PE.

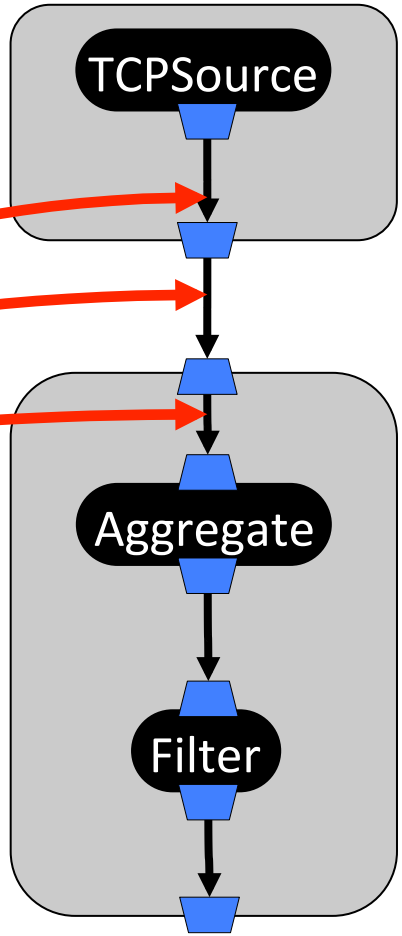
Lesson 2

Insert threaded ports into PEs to increase throughput through pipeline parallelism.

PEs and Pipeline Parallelism

We could have introduced another PE to obtain pipeline parallelism, but recall the **communication cost** across PEs:

1. tuple serialization
2. tuple transfer
3. tuple deserialization



Threaded ports do introduce a tuple copy, but it is still **less expensive** than sending tuples across PEs.

Lesson 2

Insert threaded ports into PEs to increase throughput through pipeline parallelism.

Corollary 2.1

Prefer threaded ports over PEs to obtain pipeline parallelism.

One PE, One Host

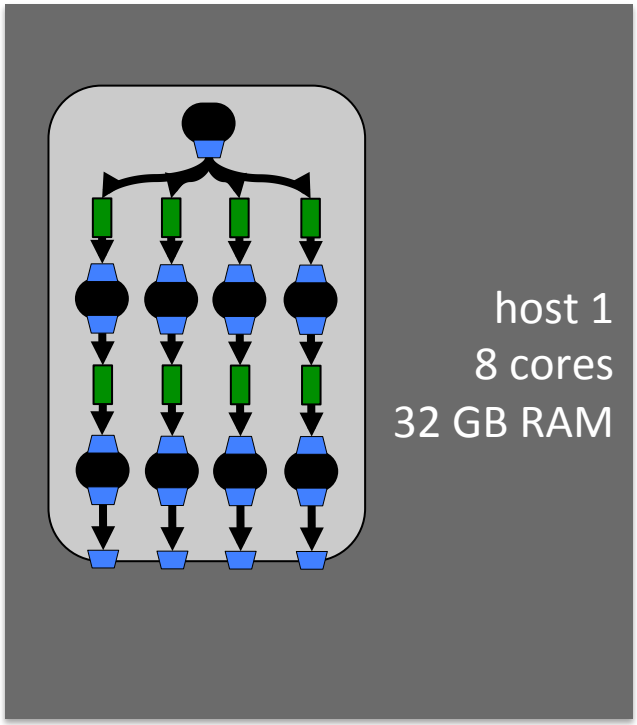
We have applied Lessons 1 and 2; we fused operators into a PE to reduce communication costs, and used threaded ports for parallelism:

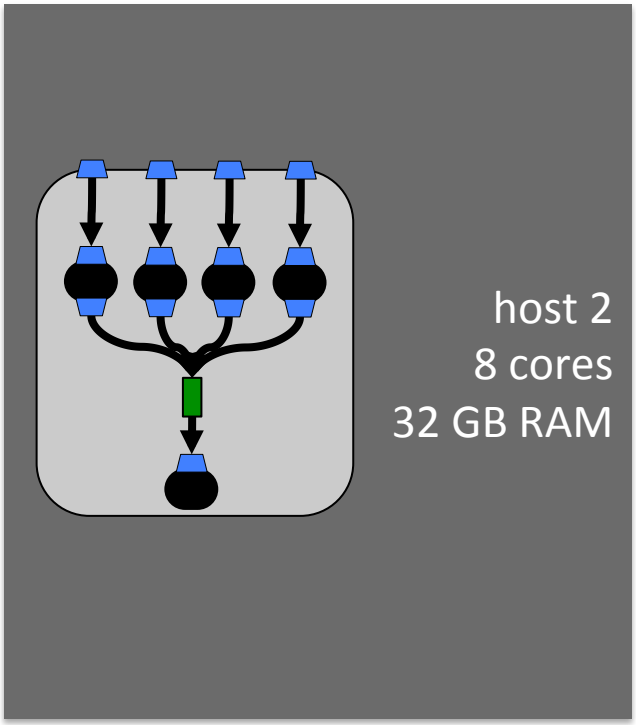


But now we have 9 threads on 8 cores, and we want to add even more threads. Our one PE is **limited to one host**. We need to spread our application to another host.

Multiple PEs, Multiple Hosts

By splitting our application into multiple PEs, we can take advantage of multiple hosts. Using multiples hosts lets us use **more resources** (cores, memory, disk, etc.) in total:



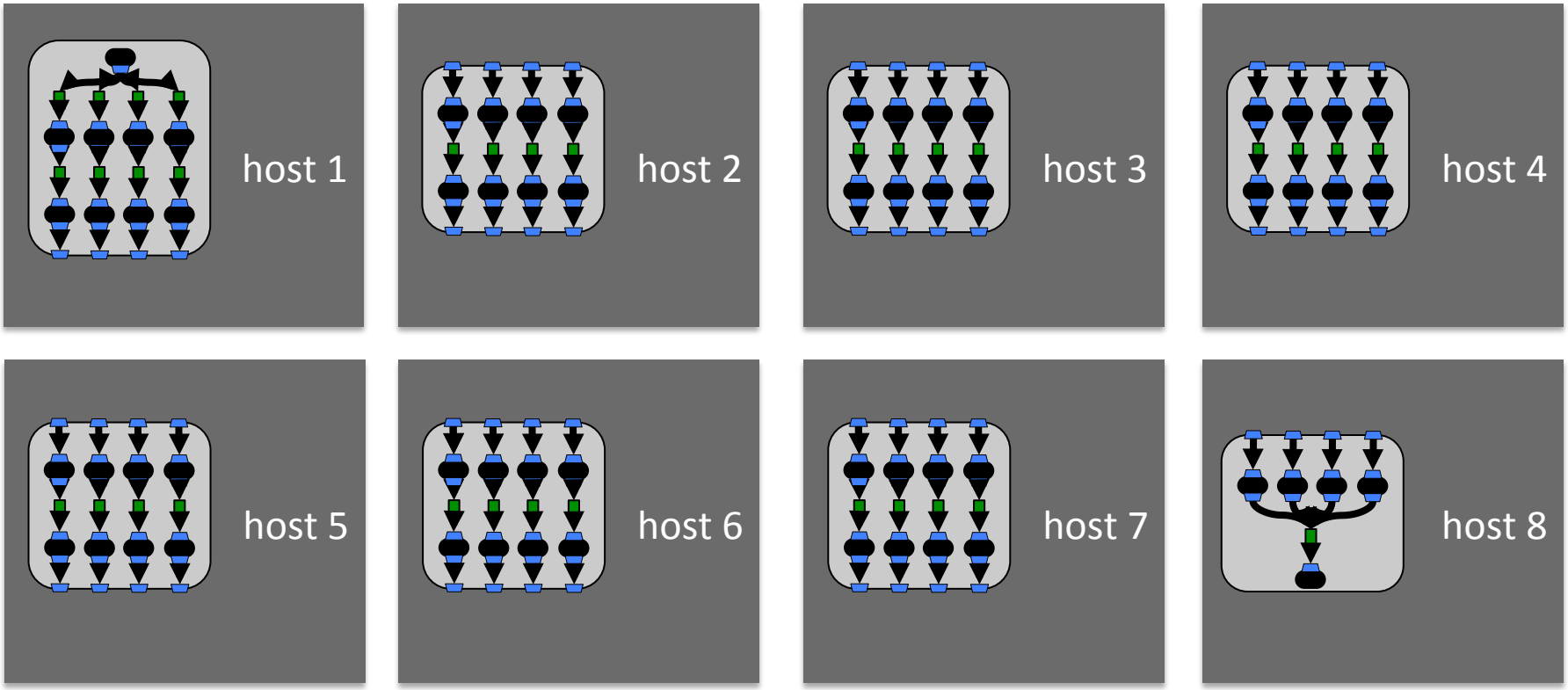


Lesson 3

Use multiple PEs in an application to take advantage of multiple hosts.

Many Hosts

Extrapolating to many hosts, the picture will tend to look like below:



If operators are on the same host, and they communicate, they should probably be **in the same PE**. We can use **threaded ports** inside the PE for pipeline parallelism.

Lesson 4

Use one PE per host.

Corollary 4.1

If there are two PEs on the same host, they should probably be fused into one PE. **Insert threaded ports** to regain parallelism.

Lesson 4, Derived

Lesson 1: fuse operators into PEs

\+ Lesson 2: insert threaded ports

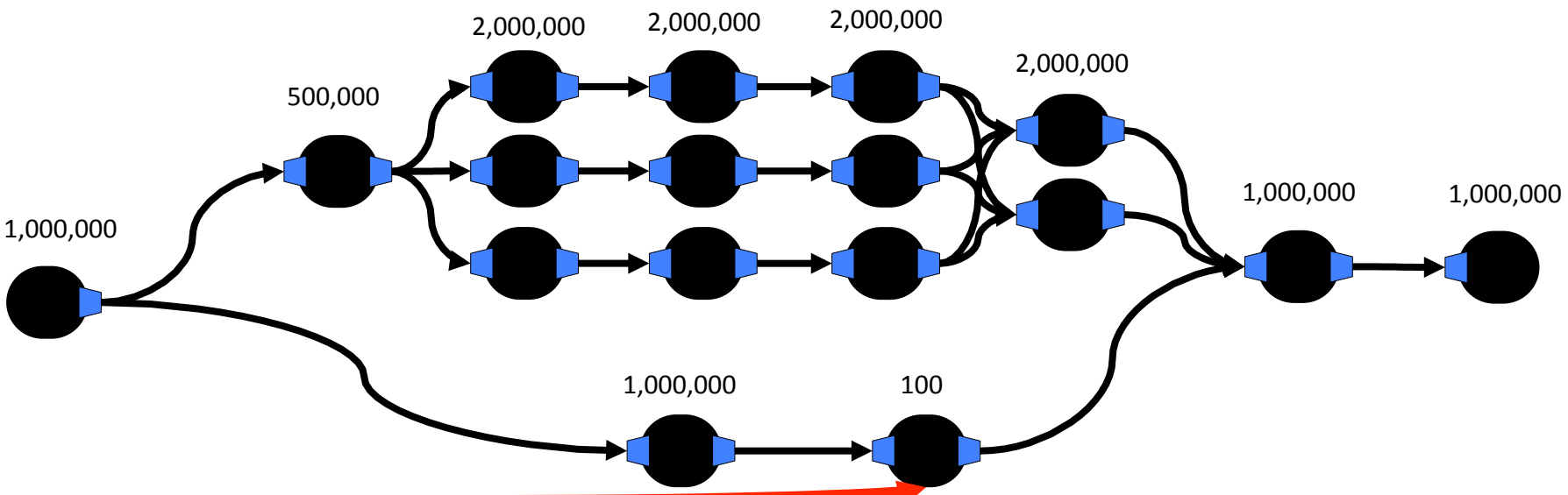
\+ Lesson 3: multiple PEs to use multiple hosts

= Lesson 4: one PE per host

Bottlenecks

All Streams applications have a **bottleneck**—an operator, or a group of operators, that are the slowest in the application. Such operators limit the overall throughput of the application.

Assume the following rates are **potential** throughputs in tuples per second:

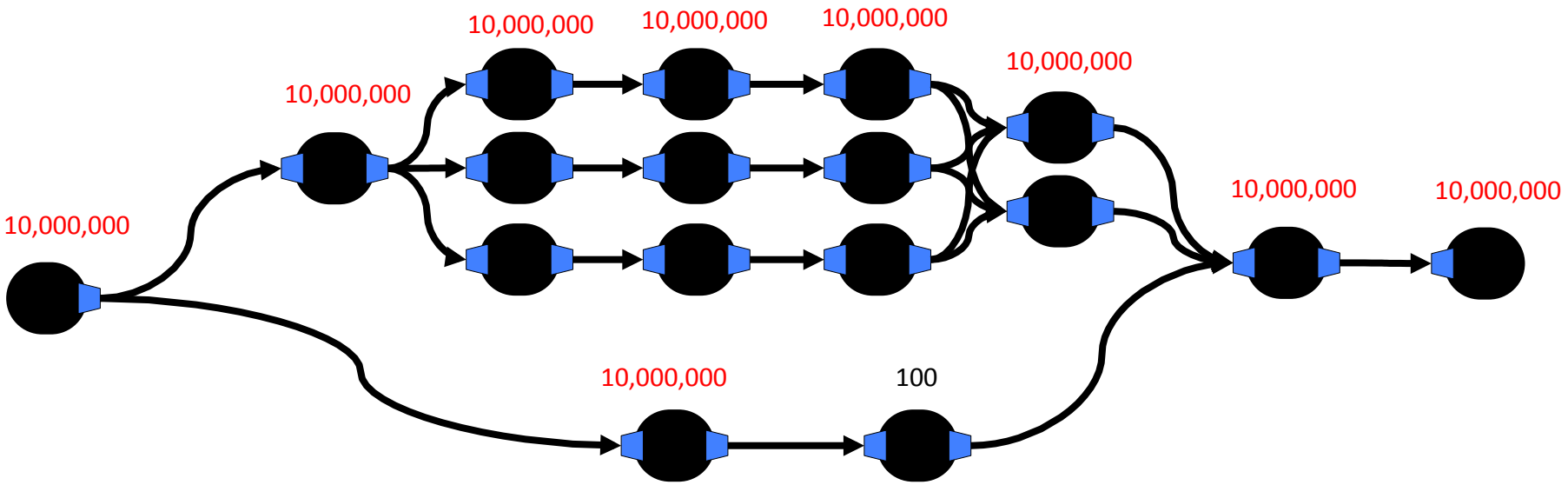


Lesson 5

Improve the performance of bottlenecks to improve the throughput of an application.

Bottlenecks are Bottlenecks

We don't like 100 tuples a second, so we improve the performance of some of our operators:



Result: **100 tuples per second**. Our bottleneck limits the throughput of the application. That's what a bottleneck **is**. We need to improve the performance of our bottleneck to improve the performance of the application.

Lesson 5

Improve the performance of bottlenecks to improve the throughput of an application.

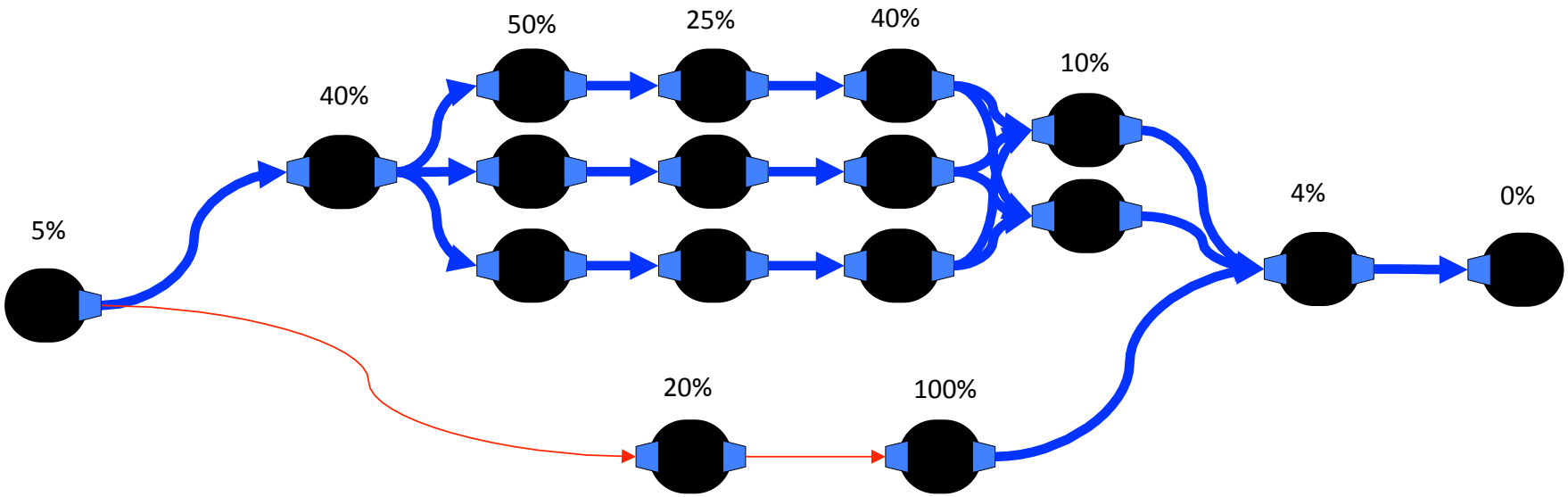
Corollary 5.1

Trying to improve the performance of an application without knowing who is the bottleneck is a **waste of time**.

Finding Bottlenecks

We can detect bottlenecks by making two systems observations:

1. Bottlenecks will likely consume close to **100% CPU**, while other operators are **mostly idle**.
2. Outgoing connections **from** bottlenecks will tend to have low congestion, but incoming connections **to** bottlenecks will tend to have high congestion.



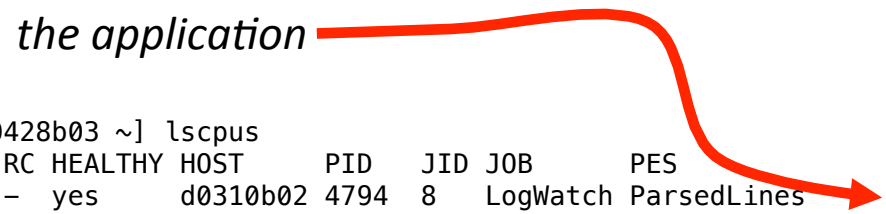
Bottleneck Detector: `lscpus`

NAME
lscpus - display CPU utlization for PEs in a Streams instance

SYNOPSIS
lscpus [**-i** instance_name] [**-s** sort_by_key] [**-t**]

DESCRIPTION
Lscpus correlates information from **streamtool lspes** and the CPU utilization from each host that those PEs are running on, as reported by **top.** If each PE contains only one operator, then the PE with the highest CPU utilization tends to be the bottleneck of the entire application. This information is useful when optimizing an application's performance.

when each PE contains one operator, the PE with the highest utilization is likely the bottleneck of the application

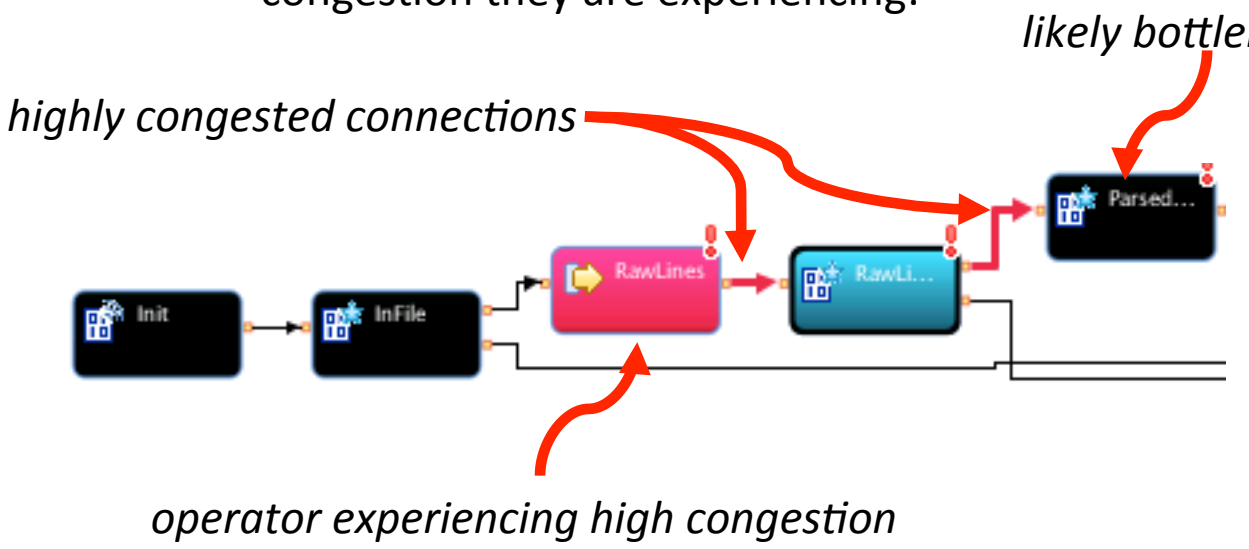


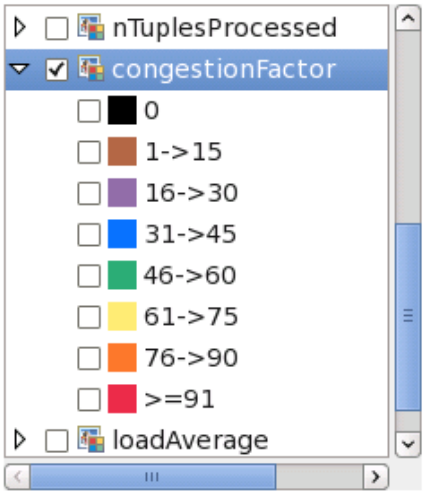
```
[scoschne@d0428b03 ~] lscpus
ID  STATE   RC HEALTHY HOST     PID   JID JOB      PES              CPU   VIRT RES SHR
129 Running -  yes     d0310b02 4794  8   LogWatch ParsedLines      102.0 679m 28m 20m
131 Running -  yes     d0310b05 20583 8   LogWatch Failures         84.6  679m 29m 20m
136 Running -  yes     d0310b05 20584 8   LogWatch RealTime.Range   61.0  682m 29m 21m
138 Running -  yes     d0310b02 4795  8   LogWatch RealTime.Diff    21.6  679m 28m 20m
130 Running -  yes     d0310b06 1313  8   LogWatch RawFailures      19.6  679m 28m 20m
137 Running -  yes     d0310b02 4796  8   LogWatch RealTime.Cutoff 19.6  753m 28m 20m
...
```

Bottleneck Detector: StreamStudio

PE connections report a **congestionFactor**, which is a measure of how often that PE has been blocked when trying to submit a tuple over the network. Values near 100 mean the connection is highly congested, and values near 0 mean the connection has negligible congestion.

StreamsStudio allows us to color operators and PEs by the congestion they are experiencing:



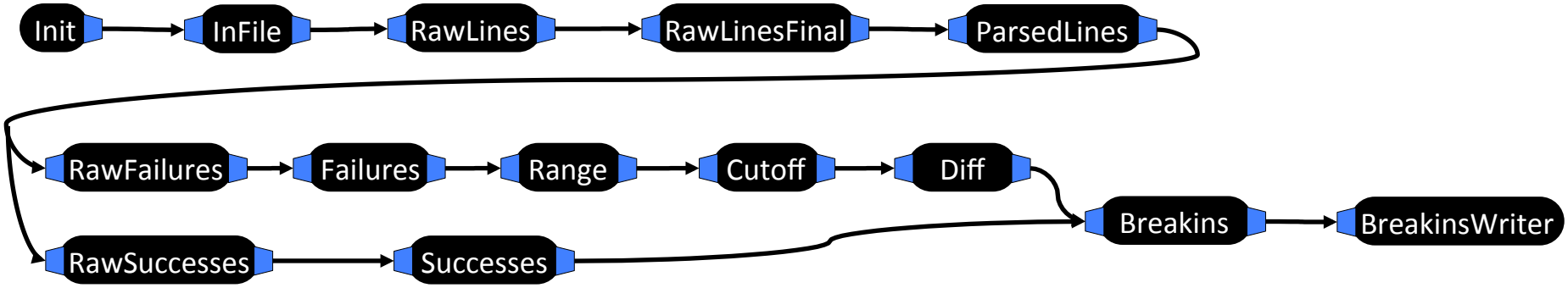


Lesson 6

Identify bottlenecks by observing CPU utilization and congestion.

Case Study: LogWatch

This is the stream graph for the sample application LogWatch:



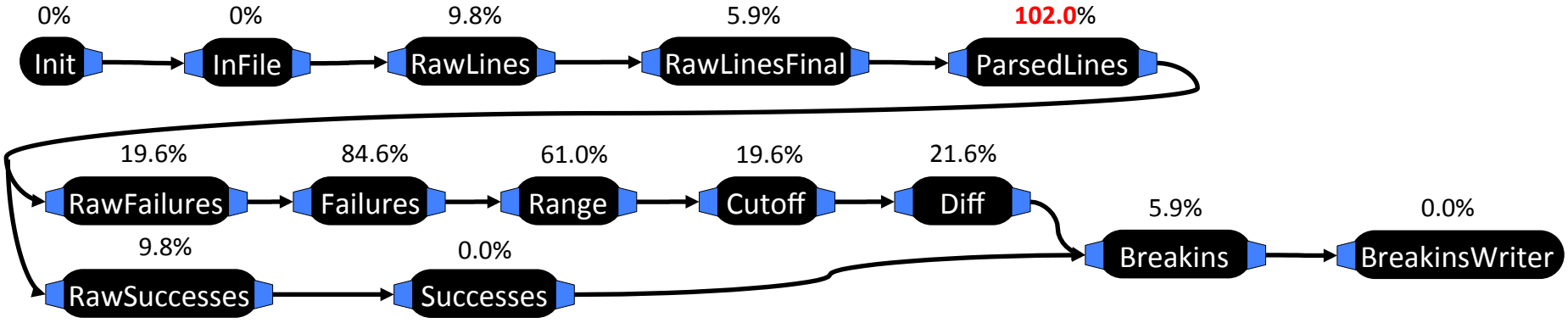
LogWatch is an application that watches the system messages file (usually found at `/var/log/messages`), flagging security breaches. Breakins are detected using the following observation:

> *If the same remote host attempts to login many times in a short period of time, then succeeds, it is likely a breakin.*

LogWatch Performance

Initial throughput: ~**49,000 tuples per second**. Let's use `lscpus` to find the bottleneck:

```
[scoschne@d0428b03 ~] lscpus
ID  STATE   RC HEALTHY HOST     PID   JID JOB      PES              CPU   VIRT RES SHR
129 Running -  yes     d0310b02 4794  8   LogWatch ParsedLines      102.0 679m 28m 20m
131 Running -  yes     d0310b05 20583 8   LogWatch Failures         84.6  679m 29m 20m
136 Running -  yes     d0310b05 20584 8   LogWatch RealTime.Range   61.0  682m 29m 21m
138 Running -  yes     d0310b02 4795  8   LogWatch RealTime.Diff    21.6  679m 28m 20m
130 Running -  yes     d0310b06 1313  8   LogWatch RawFailures      19.6  679m 28m 20m
137 Running -  yes     d0310b02 4796  8   LogWatch RealTime.Cutoff  19.6  753m 28m 20m
132 Running -  yes     d0310b06 1315  8   LogWatch RawSuccesses     9.8   679m 28m 20m
127 Running -  yes     d0310b02 4793  8   LogWatch RawLines         9.8   682m 29m 21m
134 Running -  yes     d0310b04 7946  8   LogWatch Breakins         5.9   748m 39m 21m
128 Running -  yes     d0310b02 4792  8   LogWatch RawLinesFinal    5.9   680m 28m 20m
135 Running -  yes     d0310b06 1314  8   LogWatch BreakinsWriter   0.0   681m 28m 20m
125 Running -  yes     d0310b04 7943  8   LogWatch Init             0.0   624m 28m 20m
126 Running -  yes     d0310b04 7944  8   LogWatch InFile           0.0   680m 29m 20m
124 Running -  yes     d0310b04 7945  8   LogWatch ExecTime,...     0.0   747m 28m 20m
133 Running -  yes     d0310b03 6352  8   LogWatch Successes        0.0   679m 29m 21m
```



Uncork the Bottleneck

In general, we have two options to improve the performance of a bottleneck:

1. Improve the **sequential performance** of the operator. This entails doing all of the typical optimization tricks we know from other languages.
2. Exploit **data parallelism**.

We will focus on data parallelism:



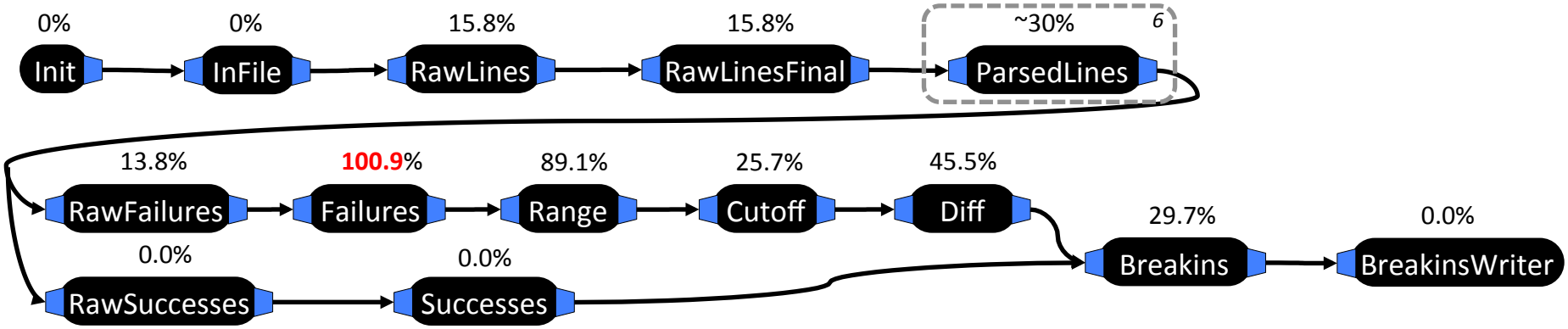
```
@parallel(width=7)
stream<LogLine> ParsedLines = Custom(RawLinesFinal) {
  logic onTuple RawLinesFinal: {
    list<rstring> tokens = tokenize(line, " ", false);
    timestamp t = timeStringToTimestamp(tokens[1] + "-" +
                                        upper(tokens[0]) + "-2011",
                                        tokens[2] + ".000", false);

                  submit({seqno = RawLinesFinal.seqno,
                          time = t, hostname = tokens[3],
                          service = tokens[4],
                          message = flatten(tokens[5:])},
                         ParsedLines);
  }
}
```

Parallelized ParsedLines

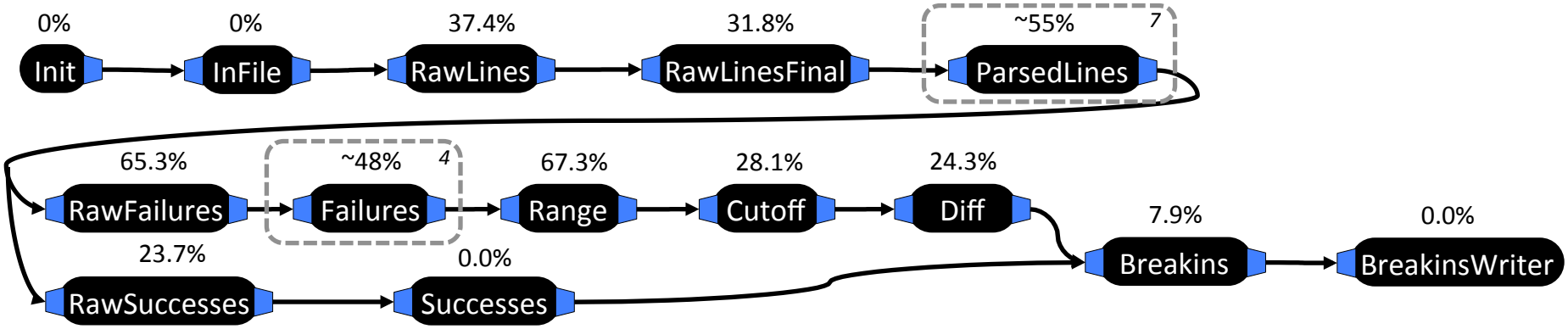
New throughput: **~96,000 tuples per second**. That is a **~2× improvement**. Let's look at the new utilizations:

```
[scoschne@d0428b03 ~] lscpus
ID  STATE   RC HEALTHY HOST     PID   JID JOB      PES                CPU   VIRT RES  SHR
147 Running -  yes     d0310b06 9914  9   LogWatch Failures           100.9 679m 31m  20m
152 Running -  yes     d0310b06 9915  9   LogWatch RealTime.Range     89.1  682m 29m  21m
154 Running -  yes     d0310b04 17756 9   LogWatch RealTime.Diff      45.5  679m 28m  20m
145 Running -  yes     d0310b05 31477 9   LogWatch ParsedLinesMerged 33.6  679m 31m  20m
155 Running -  yes     d0310b02 8140  9   LogWatch ParsedLines[1]     31.5  679m 28m  20m
150 Running -  yes     d0310b04 17757 9   LogWatch Breakins           29.7  745m 34m  21m
156 Running -  yes     d0310b02 8137  9   LogWatch ParsedLines[2]     29.5  679m 28m  20m
157 Running -  yes     d0310b02 8138  9   LogWatch ParsedLines[3]     29.5  679m 28m  20m
158 Running -  yes     d0310b02 8139  9   LogWatch ParsedLines[4]     29.5  679m 28m  20m
159 Running -  yes     d0310b02 8141  9   LogWatch ParsedLines[5]     29.5  679m 28m  20m
144 Running -  yes     d0310b02 8132  9   LogWatch ParsedLines[0]     27.6  679m 28m  20m
153 Running -  yes     d0310b04 17754 9   LogWatch RealTime.Cutoff    25.7  679m 28m  20m
142 Running -  yes     d0310b04 17752 9   LogWatch RawLines           15.8  682m 29m  21m
143 Running -  yes     d0310b04 17755 9   LogWatch RawLinesFinal      15.8  685m 31m  20m
146 Running -  yes     d0310b03 17918 9   LogWatch RawFailures        13.8  679m 28m  20m
```



Parallelized `Failures`

New throughput: **~215,000 tuples per second**. That is a **~4× improvement**. The raw utilizations from `lscpus` are crowded now (lots of parallel operators), so we'll just look at the figure:



Looking at the utilizations for `RawFailures` and `Range`, it looks like we can probably squeeze some more data parallel performance from this application. **But.** I inserted (unshown) merger operators after the parallel regions, and they have become the bottleneck, at **89%** and **99%** utilization.

Lesson 5

Improve the performance of bottlenecks to improve the throughput of an application.

Corollary 5.1

Trying to improve the performance of an application without knowing who is the bottleneck is a **waste of time**.

Corollary 5.2

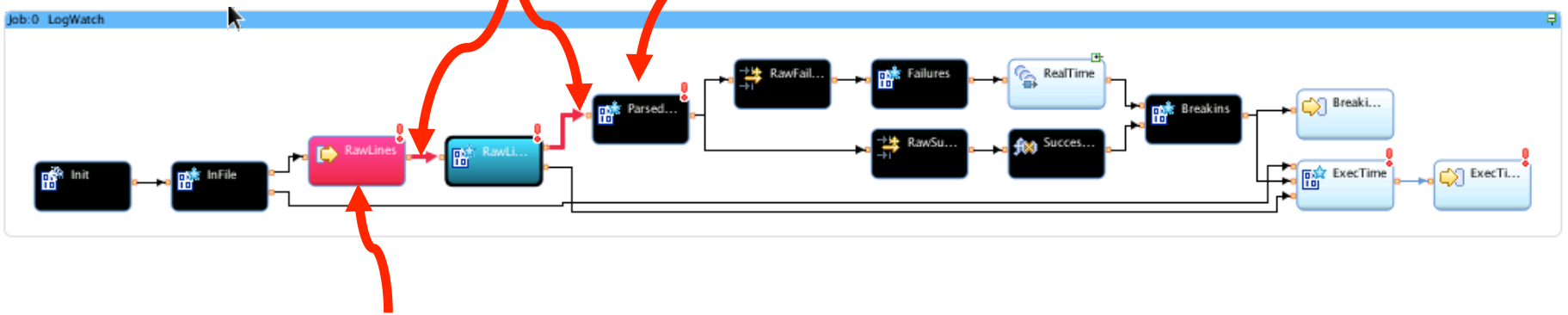
When a parallel region is no longer the bottleneck, further parallelism **will not help**.

StreamStudio and Congestion

We also could have identified bottlenecks by using StreamStudio and looking at congestion. The original application:

`ParsedLines`, *which is our bottleneck*

highly congested connections



operator experiencing congestion

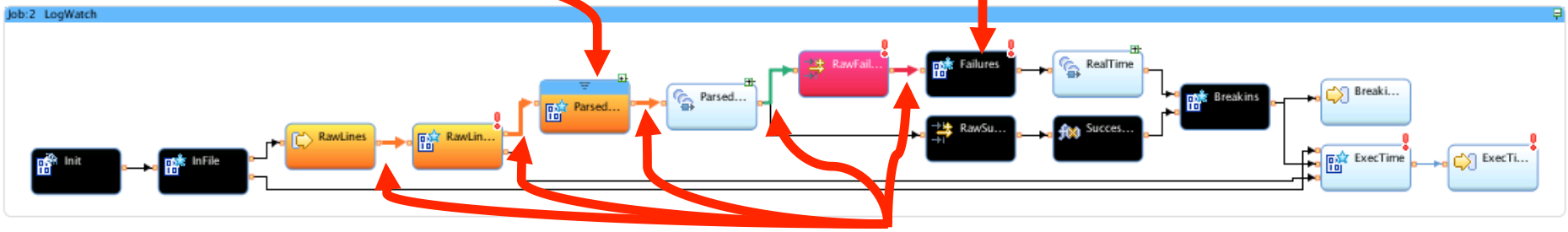
Note that operators **downstream** of the bottleneck experience no congestion, but operators **upstream** of the bottleneck experience congestion. That, in fact, is how we identify the bottleneck.

Congestion as We Add Parallelism

We will follow the same process we did when identifying bottlenecks with CPU utlization. The congestion view after applying UDP to `ParsedLines`:

we parallelized `ParsedLines`*, so it is no longer the bottleneck*

`Failures` *is the new bottleneck*

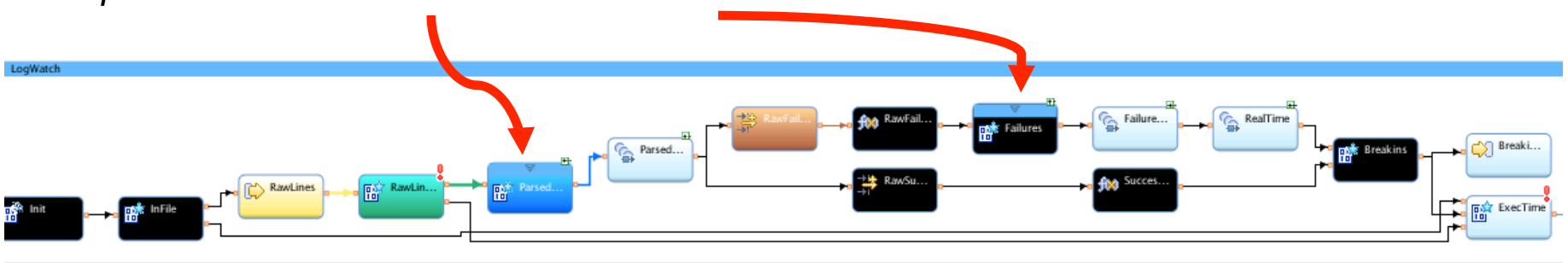


again, note that back-pressure causes connections upstream from the bottleneck to be highly congested

Relatively Balanced Congestion

After we apply UDP to `Failures`, the congestion view shows improvement:

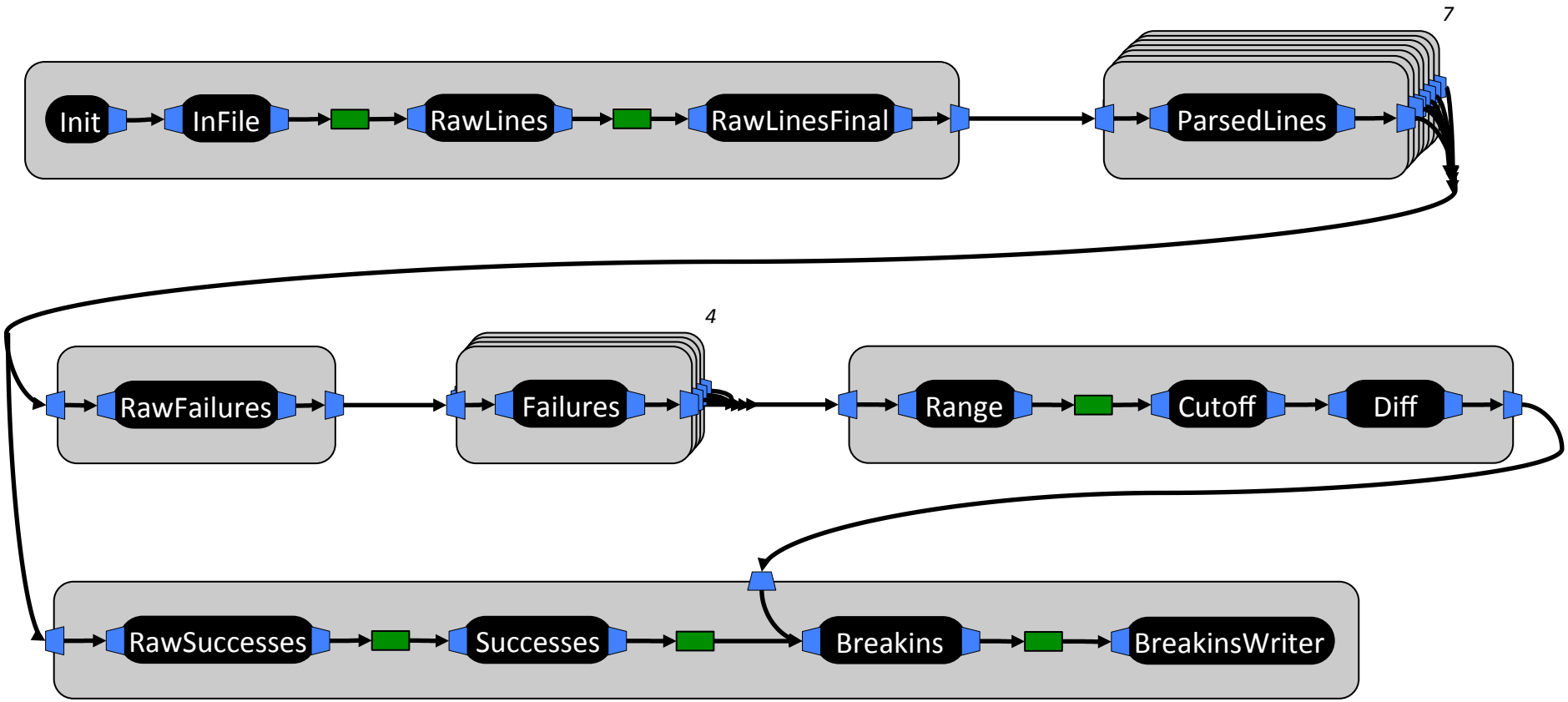
we parallelized both `ParsedLines` *and* `Failures`



There is no longer any obvious bottleneck from congestion—that is what we want, as it means the relative costs of the operators are **balanced**.

When looking for bottlenecks using either CPU utilization or congestion, we want to end up with a balanced system: no one operator uses much more CPU than another, and no one operator causes much more congestion than another.

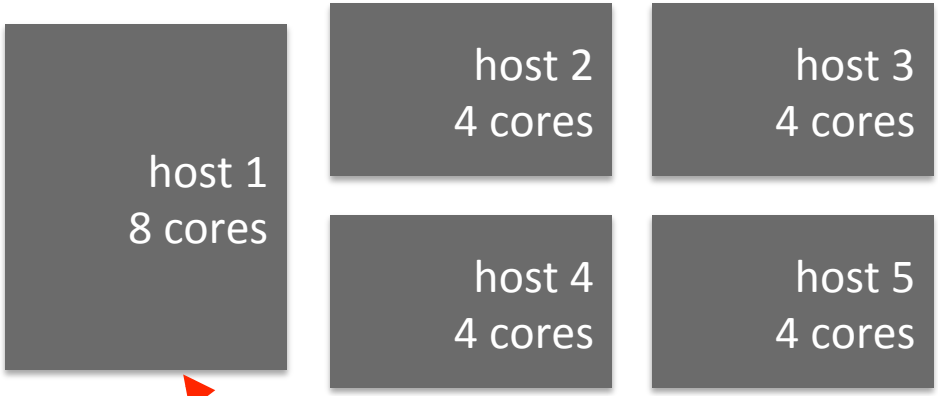
Applying the Other Lessons



We have fused operators into PEs, and inserted threaded ports where appropriate. But, we still need to consider host placement; where these PEs execute can make a big difference in performance.

Host Assignment

Our instance has five hosts and a total of 24 cores. Our application is processing-limited, so we want to place PEs to maximize core usage.




```
@parallel(width=7)
stream<LogLine> ParsedLines = Custom(RawLinesFinal) {
  logic onTuple RawLinesFinal: {
    list<rstring> tokens = tokenize(line, " ", false);
    timestamp t = timeStringToTimestamp(tokens[1] + "-" +
                                        upper(tokens[0]) + "-2011",
                                        tokens[2] + ".000", false);

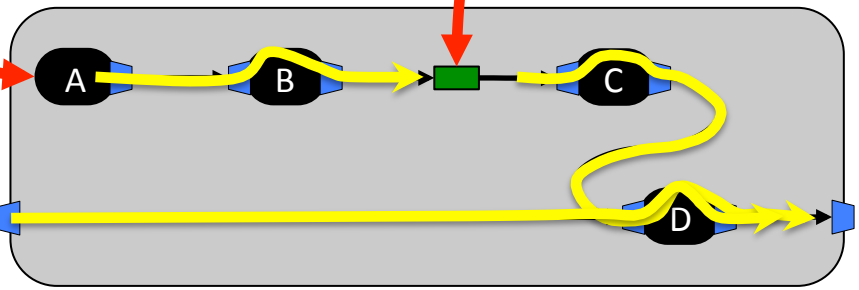
                  submit({seqno = RawLinesFinal.seqno,
                          time = t, hostname = tokens[3],
                          service = tokens[4],
                          message = flatten(tokens[5:])},
                         ParsedLines);
  }
  config placement: host(nodes[0]);
}
```

Detour: Sources of Threads

Threads execute all downstream operators until they encounter a threaded port, or a PE output port.

source operators are driven by their own thread.

threaded ports introduce a thread (hence the name)

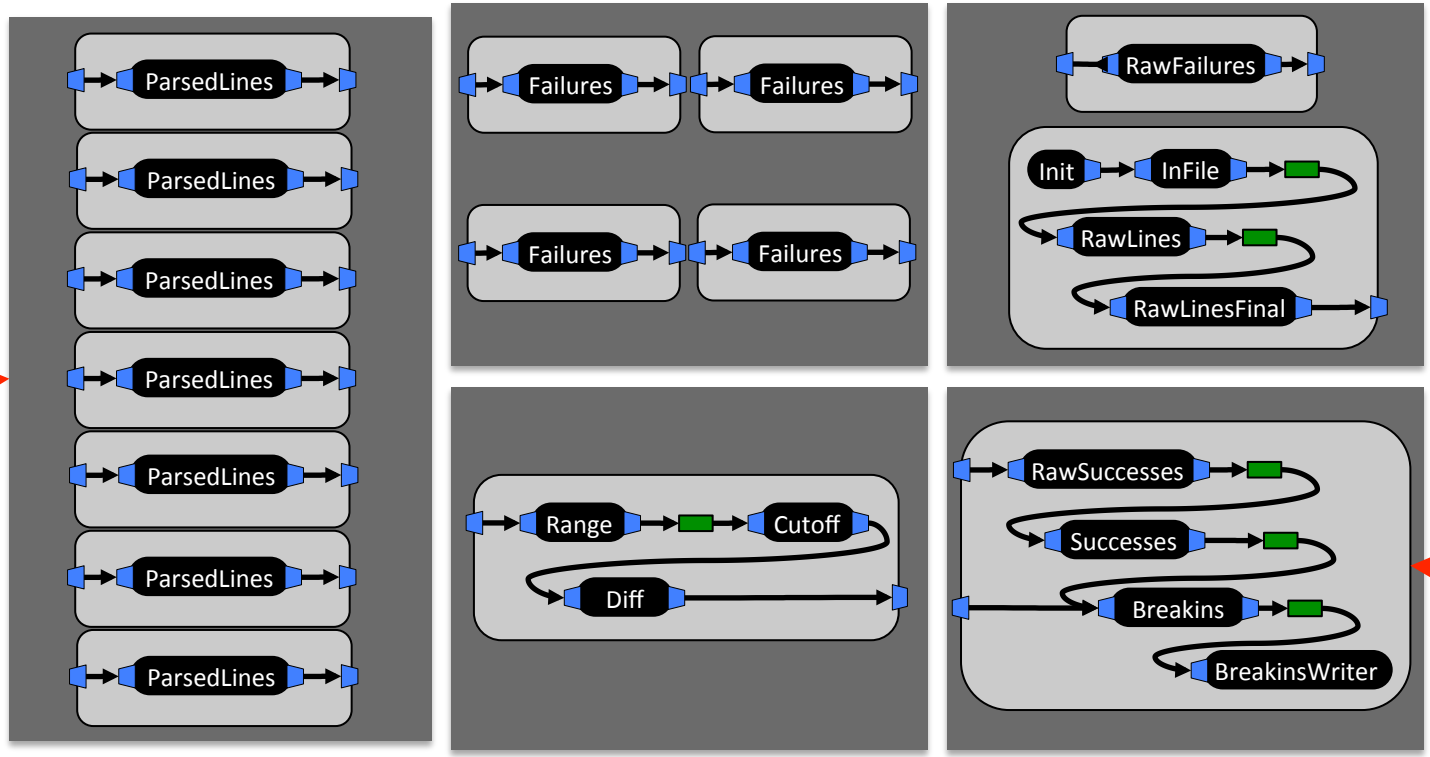


PE input ports introduce a thread which pulls tuples from the network and executes downstream operators.

Operators can also have **their own threads**. What are shown above are the threads introduced by the system itself.

Full Host Placement

These PE assignments makes sure that no host has more threads than it can handle:



*8 cores in the host, 7 PEs and **7 threads total***

*4 cores in the host, 1 PE and **5 threads total**; more threads than cores, but recall that the utilization of these operators is relatively low*

Case Study: Final Results

Initial throughput:

~49,000 tuples per second

Parallelized first bottleneck:

~**96,000** tuples per second, ~**2×** improvement

Parallelized second bottleneck:

~**215,000** tuples per second, ~**4×** improvement

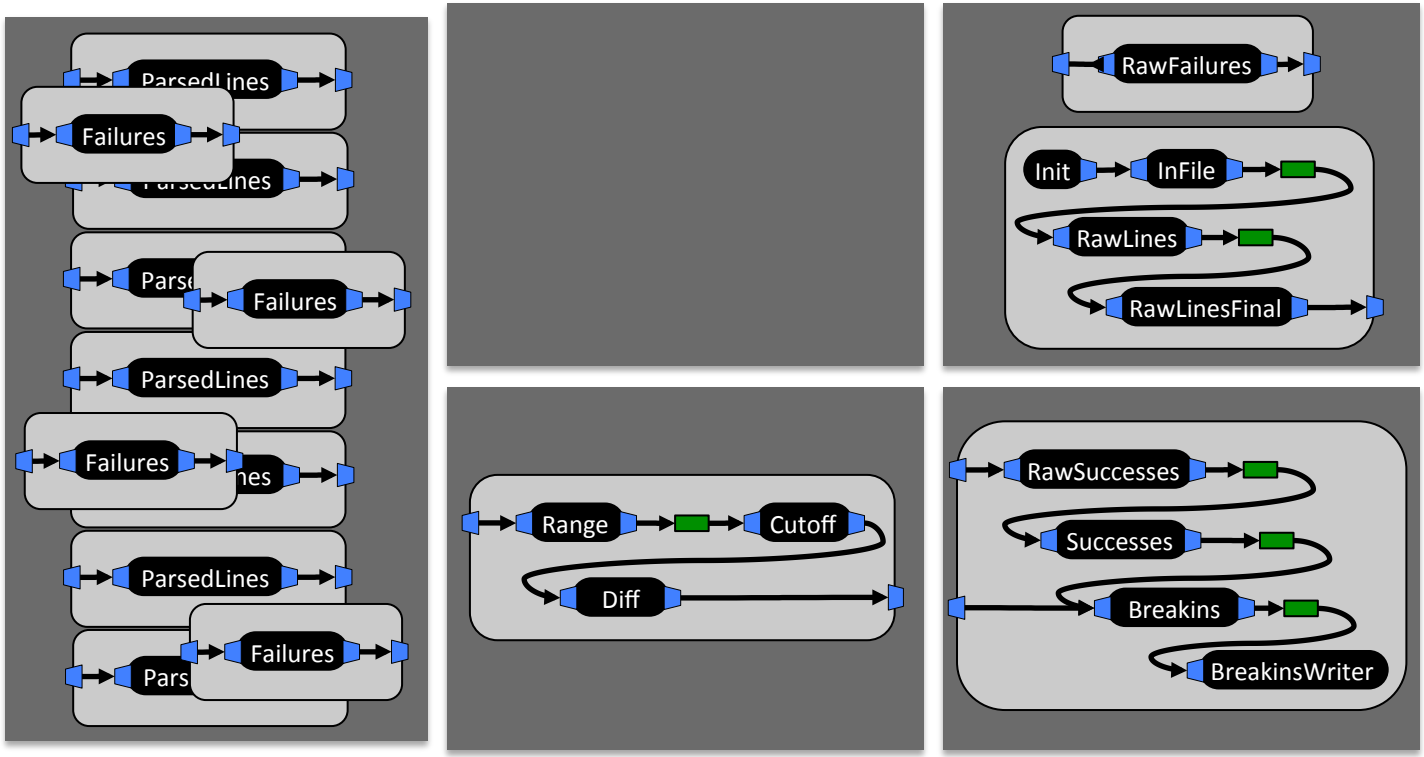
Fused operators into PEs, inserted threaded ports, assigned to hosts:

~**215,000** tuples per second, ~**4×** improvement

Fusion actually does not make a difference in the overall throughput for this application. Why? The tuple size is relatively small, so the transfer costs were already small compared to other costs.

Bad Placement

Throughput: **105,000** tuples per second. That is a **2× drop** in performance, from ~210,000, just because of placement.



Visualized like this, it is obviously a bad placement. Which means you need to **know your hardware**.

Lesson 6

Know your hardware. Distribute PEs to hosts so as to avoid over-subscribing any resource (cores, memory, disk, etc.) on that host.

Remember Lesson 0?

`sc –a –M LogWatch`

~215,000 tuples per second

`sc –M LogWatch`

~110,000 tuples per second

Both numbers represent runs with the same operator fusion, threaded port placement, and host assignments. The only difference is `–a`.

The `–a` option turns on optimizations. The SPL compiler itself will perform optimizations on the code it generates, and it will instruct `g++` to optimize at level `–O3` for the generated C++.

Without `–a`, the SPL compiler will perform no optimizations, and will not instruct `g++` to optimize the generated C++.

If you care about performance, **compile with `–a`**.

Lessons

0. Compile with `-a`.
1. Fuse operators into the same PE to reduce communication costs.
2. Insert threaded ports into PEs to increase throughput through pipeline parallelism.
    1. Prefer threaded ports over PEs to obtain pipeline parallelism.
3. Use multiple PEs in an application to take advantage of multiple hosts.
4. Use one PE per host.
    1. If there are two PEs on the same host, they should probably be fused into one PE. **Insert threaded ports** to regain parallelism.
5. Improve the performance of bottlenecks to improve the throughput of an application.
    1. Trying to improve the performance of an application without knowing who is the bottleneck is a **waste of time**.
    2. When a parallel region is no longer the bottleneck, further parallelism **will not help**.
6. Know your hardware. Distribute PEs to hosts so as to avoid over-subscribing any resource (cores, memory, disk, etc.) on that host.

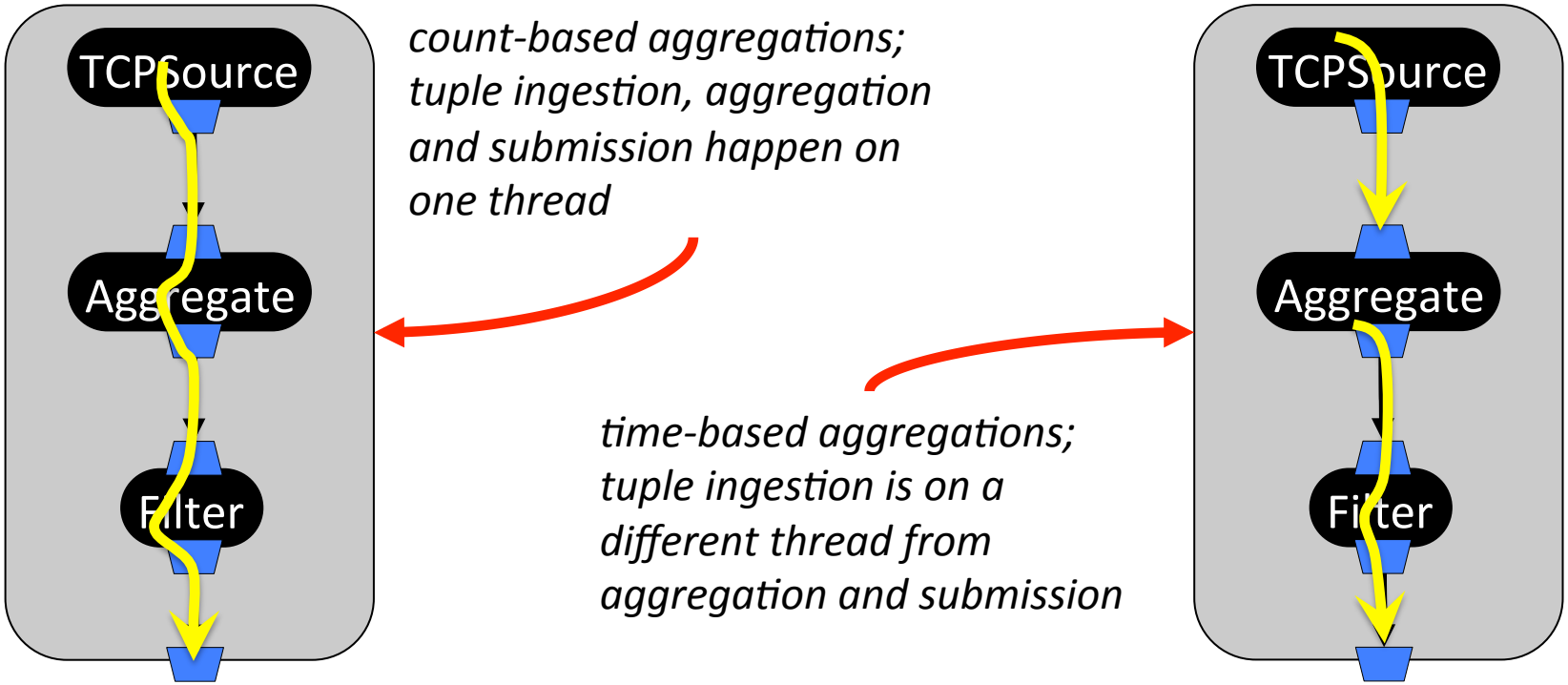
"Education is a series of small lies."

—*my sophomore year Intro to Computer Engineering professor, explaining that, in fact, some real computer hardware uses ternary, not binary, logic*

I have "omitted some details" (lied) in order to help the flow of the lessons, so in the next few slides we will correct that.

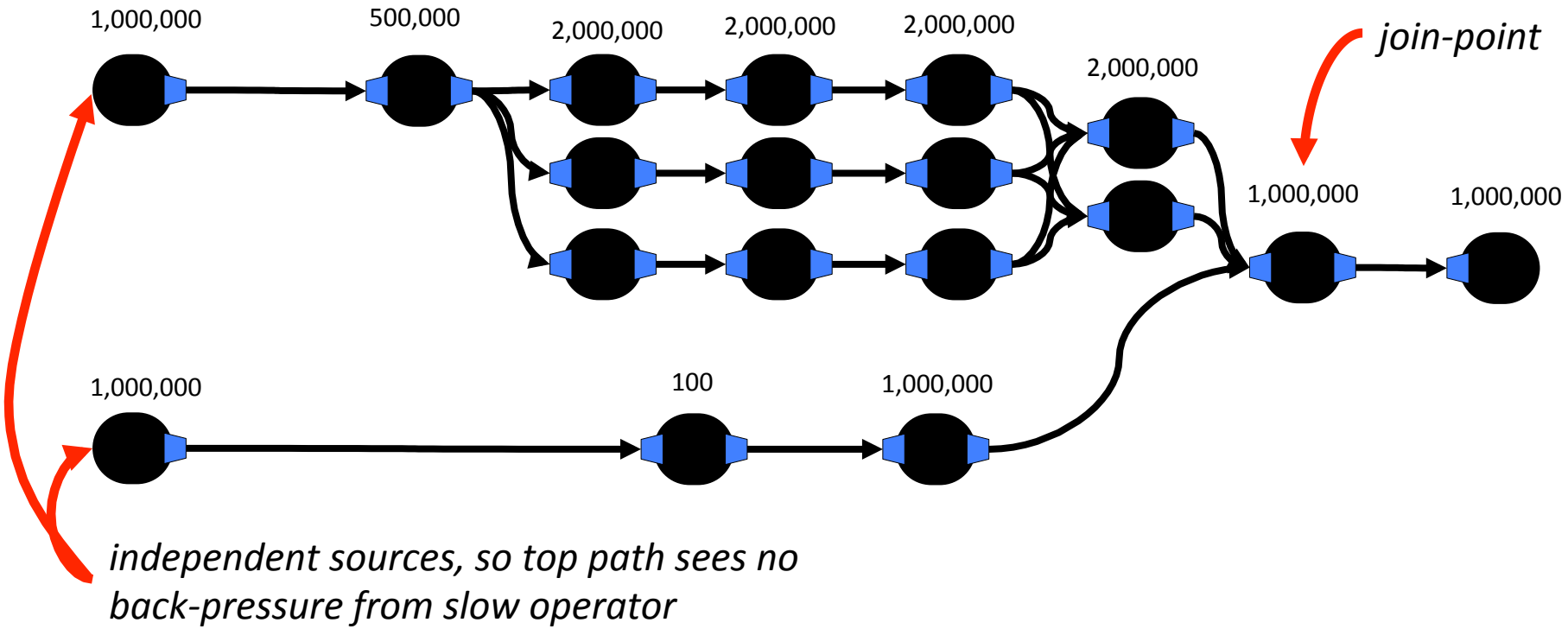
Time-Based Aggregate

The thread-execution picture on the left is not always accurate. The `Aggregate` operator allows time-based windows. After a certain number of seconds, the operator performs an aggregation and produces a tuple. As tuple production happens based on a timer, and is not initiated by ingesting a tuple, the operator uses another thread:



Not Necessarily a Bottleneck, Part 1

In the picture below, the throughput is **not necessarily** 100 tuples per second. If the join-point (labeled) is not a barrier—that is, if that operator does **not need to wait** for tuples from all input ports—then receiving only 100 tuples per second on one input port may not slow it down.



Not Necessarily a Bottleneck, Part 2

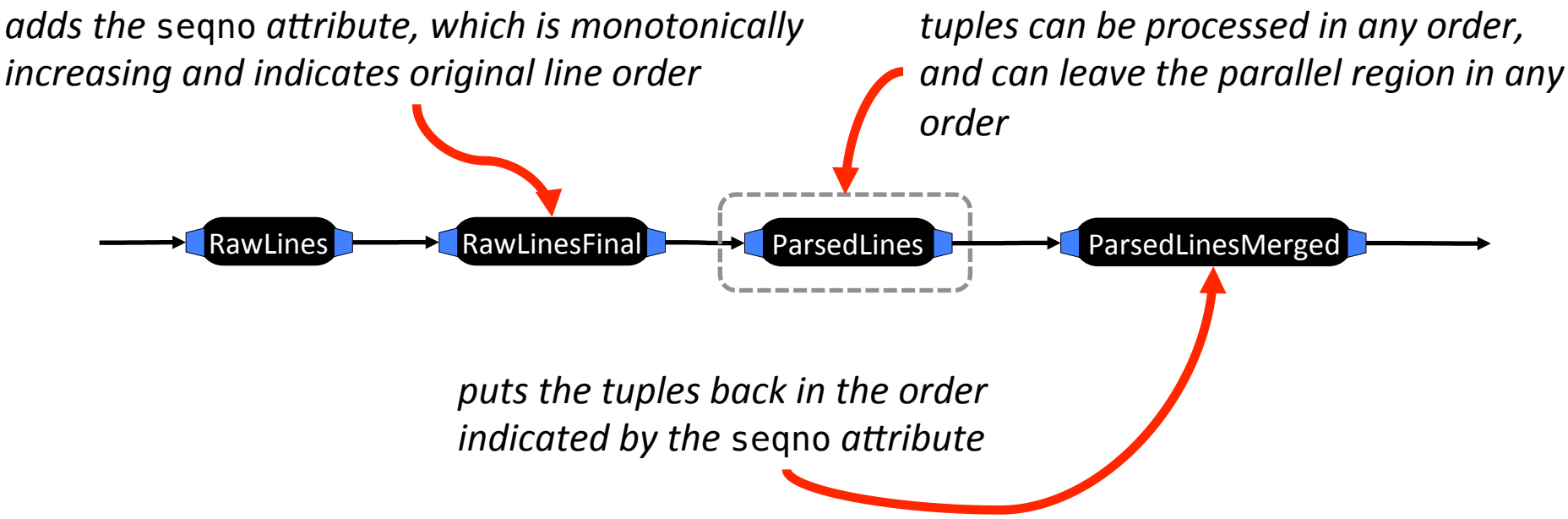
Previous bottleneck discussions talked about **potential throughputs**, but an important detail about **actual throughputs** (that is, throughputs as seen in a running application) is **selectivity**. If an operator is highly selective, it may be normal for it to have low throughput. Selectivity is the number of tuples produced for tuples consumed:

	1:[0,1]	1:1	1:∞
	operator produces either 0 or 1 tuple for every tuple consumed; highly selective operators are more often 0	operator produces 1 tuple for every tuple consumed	operator produces more than 1 tuple, potentially many, for every tuple consumed
examples	`Filter`, `Aggregate`, `DeDuplicate`	`Functor` w/o `filter`, `Parse`	`XMLParse`, `Aggregate` w/ `groupBy`

In addition, operators **with their own threads** can generate a tuple at any time, independent of receiving an input tuple. An example is a time-based `Aggregate`.

UDP and Ordering

When looking for suspects in LogWatch, we actually depend on the tuple order being the same as it was in the log file. UDP does not preserve tuple order. I added an explicit merge to put tuples back into order after the parallel region for `ParsedLines`:



Ordering: The Code

The merge is straight-forward, and depends on the fact that the parallel region is 1:1—that is, for every tuple that enters the parallel region, one tuple leaves it. No tuples are "lost" or spuriously created.

```
composite UDPMerge(input In; output Out) {
param attribute $key;

graph
  stream<In> Out = Custom(In) {
    logic state: {
      mutable map<uint64, tuple<In>> _tuples;
      mutable uint64 _next = 1;
    }

    onTuple In: {
      if (_next == $key) {
        submit(In, Out);
        ++_next;
        while (_next in _tuples) {
          submit(_tuples[_next], Out);
          removeM(_tuples, _next);
          ++_next;
        }
      }
      else {
        _tuples[$key] = In;
      }
    }
  }
}
```

```
@parallel(width=7)
stream<LogLine> ParsedLines = Custom(RawLinesFinal) {
  logic onTuple RawLinesFinal: {
    list<rstring> tokens = tokenize(line, " ", false);
    timestamp t = timeStringToTimestamp(tokens[1] + "-" +
                    upper(tokens[0]) + "-2011",
                    tokens[2] + ".000", false);

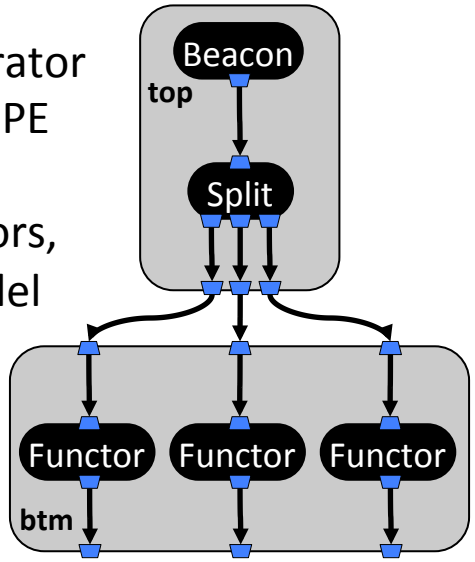
    submit({seqno = RawLinesFinal.seqno,
            time = t, hostname = tokens[3], service = tokens[4],
            message = flatten(tokens[5:])},
           ParsedLines);
  }
}

stream<LogLine> ParsedLinesMerged = UDPMerge(ParsedLines) {
  param key: ParsedLines.seqno;
}
```

Pre-UDP Data Parallelism

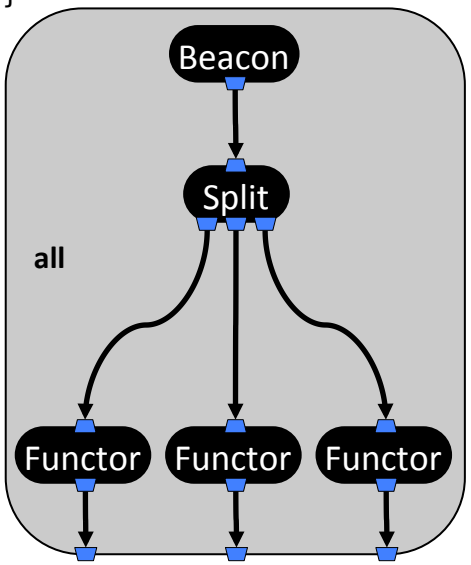
```
stream<Data> Foo = Beacon() {
  output Foo: x=random();
  config placement: partitionColocation("top");
}
(stream<Data> Foo0;
 stream<Data> Foo1;
 stream<Data> Foo2) = Split(Foo) {
  param index: hashCode(x);
  config placement: partitionColocation("top");
}
stream<Data> Bar0 = Functor(Foo0) {
  config placement: partitionColocation("btm");
}
stream<Data> Bar1 = Functor(Foo1) {
  config placement: partitionColocation("btm");
}
stream<Data> Bar2 = Functor(Foo2) {
  config placement: partitionColocation("btm");
}
```

The `Split` operator is in a different PE from the data-parallel operators, so we get parallel execution.





```
stream<Data> Foo = Beacon() {
  output Foo: x=random();
  config placement: partitionColocation("all");
}
(stream<Data> Foo0;
 stream<Data> Foo1;
 stream<Data> Foo2) = Split(Foo) {
  param index: hashCode(x);
  config placement: partitionColocation("all");
}
stream<Data> Bar0 = Functor(Foo0) {
  config placement: partitionColocation("all");
}
stream<Data> Bar1 = Functor(Foo1) {
  config placement: partitionColocation("all");
}
stream<Data> Bar2 = Functor(Foo2) {
  config placement: partitionColocation("all");
}
```



The `Split` operator is in the same PE as the data-parallel operators, with no extra threads, so we **do not get parallel execution**.

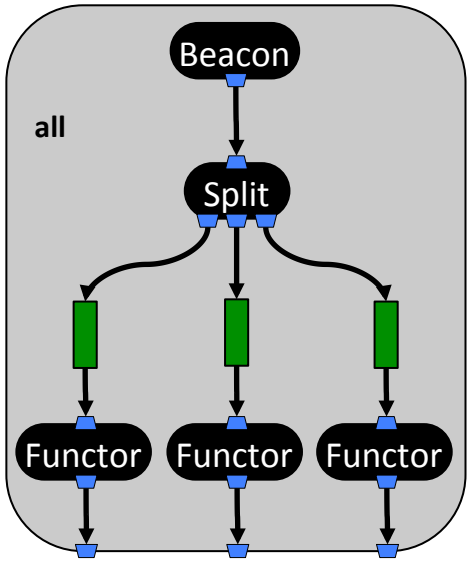


Correct Usage of `Split` in Same PE

```
stream<Data> Foo = Beacon() {
  output Foo: x=random();
  config placement: partitionColocation("all");
}

(stream<Data> Foo0;
 stream<Data> Foo1;
 stream<Data> Foo2) = Split(Foo) {
 param index: hashCode(x);
 config placement: partitionColocation("all");
}

stream<Data> Bar0 = Functor(Foo0) {
 config placement: partitionColocation("all");
        threadedPort: queue(Foo0, Sys.Wait);
}
stream<Data> Bar1 = Functor(Foo1) {
 config placement: partitionColocation("all");
        threadedPort: queue(Foo1, Sys.Wait);
}
stream<Data> Bar2 = Functor(Foo2) {
 config placement: partitionColocation("all");
        threadedPort: queue(Foo2, Sys.Wait);
}
```



The `Split` operator is in the same PE as the data-parallel operators, and there are threaded ports between them. Now the `Split` operator and the data-parallel instances **will execute in parallel**.

Correct ThreadedSplit Usage

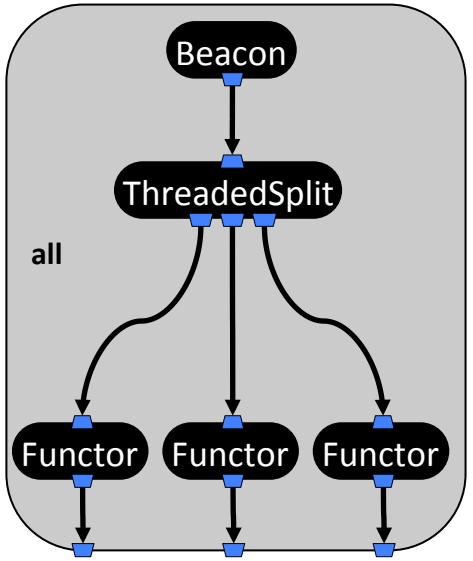
```
stream<Data> Foo = Beacon() {
  config placement: partitionColocation("all");
}

(stream<Data> Foo0;
 stream<Data> Foo1;
 stream<Data> Foo2) = ThreadedSplit(Foo) {
  config placement: partitionColocation("all");
}

stream<Data> Bar0 = Functor(Foo0) {
 config placement: partitionColocation("all");
}

stream<Data> Bar1 = Functor(Foo1) {
 config placement: partitionColocation("all");
}

stream<Data> Bar2 = Functor(Foo2) {
 config placement: partitionColocation("all");
}
```



The ThreadedSplit operator contains its own threads and queues inside of it:

